AeroUN – Sistema de reserva de tiquetes

Mitchell Elizabeth Rodríguez Barreto, Óscar Álvarez Rodríguez, John Alejandro Hernández Mora

No. grupo del curso: 1 y No. de Equipo de Trabajo: 3

I. INTRODUCCIÓN

En la actualidad, uno de los problemas más apremiantes al que se enfrentan las personas en su cotidianidad, es el manejo de la información en cualquier ámbito. Ya sea para trabajo, viajes, transporte público e incluso comidas, existe tal cantidad de información que se hace indispensable tener métodos y herramientas que faciliten su organización y análisis. Al respecto, las estructuras de datos, permitirán lograr este objetivo ya que, en esencia, son formas particulares de organizar datos de manera eficiente. De hecho, sin el estudio y aplicación apropiada de estas estructuras, empresas como Google y Facebook no podrían almacenar, filtrar y utilizar los billones de datos que diariamente deben manejar.

Teniendo en cuenta lo anterior, en el presente documento se expone el desarrollo de una aplicación de Java con ayuda de GitHub la cual permitirá acceder a la información de vuelos y reservar o buscar de acuerdo a un usuario creado. Para esto se hará uso de las estructuras de datos vistas en clase como listas doblemente enlazadas y árboles. Además se provee una interfaz desde la cual el usuario puede interactuar fácilmente con la aplicación.

I. DESCRIPCIÓN DEL PROBLEMA

Considerando lo anterior, se optó por trabajar el problema de reservar tiquetes aéreos, ya que se ve la necesidad por parte de los viajeros en elegir la opción más económica o más rápida de llegar a su destino. En particular, se quiere que la persona pueda planear su viaje de manera eficiente pensando en tiempos, costos, fechas y cupo en los vuelos.

II. USUARIOS DEL PRODUCTO DE SOFTWARE

Este software está diseñado para operar únicamente con el rol de viajero. Es decir, que está diseñado para que cualquier persona pueda ingresar al sitio web, buscar los vuelos de su interés dependiendo del origen y destino y reservar. Para mayor facilidad, se le pedirá al usuario un nombre y un correo que servirán de referencia para almacenar y cancelar las reservas.

III. REQUERIMIENTOS FUNCIONALES DEL SOFTWARE

Los requerimientos funcionales de un sistema, son aquellos que describen cualquier actividad que este deba realizar, en otras palabras, el comportamiento o función particular de un sistema o software cuando se cumplen ciertas condiciones.

- Descripciones de los datos a ser ingresados en el sistema:
  - Nombre y correo: Son los datos a partir de los cuales se caracterizará a cada usuario para asignarles una reserva específica.
  - Código por ciudad y precio: Se debe establecer un número entero que identifique cada ciudad de origen o destino y asignarle un respectivo precio. En este caso, se definió un número fijo de 9 ciudades
  - Hora del vuelo: Se debe especificar las horas de salida de los vuelos en un arreglo de 2 enteros de hora y minuto.
  - Número de sillas: Se tendrá en consideración un número fijo de sillas (30) para todos los vuelos y se establecerá un vector booleano del tamaño del número de sillas que empezará en falso.
  - Ciudades: Para realizar la búsqueda el usuario debe ingresar la fecha en que desea viajar y la ciudad de destino.

- Descripciones de las operaciones a ser realizadas en cada pantalla que se presenta:
  - En la primera pantalla, existen dos opciones, por un lado se realiza la selección de fecha, ciudad de origen y destino y se realiza la búsqueda de los vuelos que cumplan con estos valores de entrada. Por otro lado, se puede realizar la operación de ver reservas al ingresar nombre y correo.
  - En la segunda pantalla, se hace la operación de filtrado de acuerdo a precio, disponibilidad de sillas y hora. Luego, se puede escoger reservar o regresar a la pantalla anterior.
  - En la tercera pantalla, se hace la operación de mostrar la reserva elegida y se tiene la posibilidad de cancelarla y borrar éstos datos.

- Descripción de los flujos de trabajo realizados por el sistema.

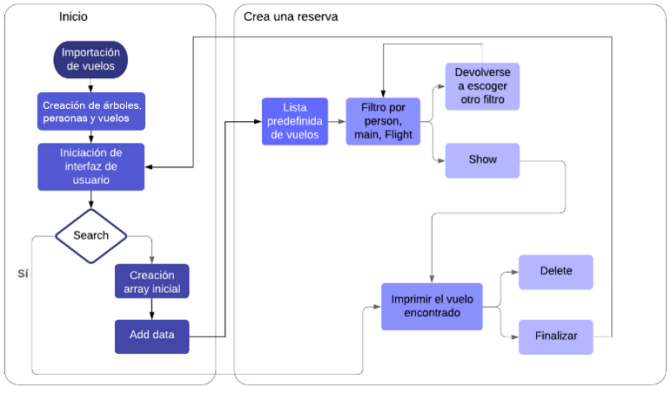


Figura 1. Diagrama de flujo del sistema. Ver Anexo A

El sistema al iniciarse, importará datos de un documento en el cual está guardada la información de los vuelos. Luego, al

momento de realizarse creará dos listas que se utilizarán durante toda la ejecución del programa, la primera será para almacenar los vuelos, la segunda para almacenar a las personas que se ingresen. Ésta última aumentará su tamaño al registrarse y traerá al inicio un número de personas predefinidas. A continuación, se iniciará la interfaz de usuario para realizar la validación, por un lado si el usuario desea ver su vuelo, puede filtrar su vuelo y dará opción de editarlo borrarlo o regresar. Si por el contrario, desea reservarlo, se creará un array que almacenará los datos, se le pedirá al usuario el dato mediante el cual desea filtrar y se le mostrará, por último, que los datos son correctos. El último paso, será crear la reserva y verla en la pantalla de información de vuelo, que dará la opción de editar, eliminar o regresar al inicio.

- Descripción de los reportes del sistema y otras salidas:
  - El reporte principal del sistema es la reserva que corresponde a cada usuario, en ella se especifica el número de vuelo, ciudad de destino, horas y costo.
  - Recordando que se quiere medir la eficiencia de las estructuras de datos usadas, se creó otra clase llamada Time que tiene como función medir el tiempo de operación. Este reporte se hará en modo comparativo y se dará una respuesta en milisegundos.

- Definición de quiénes pueden ingresar datos en el sistema:
  - Principalmente el viajero o pasajero será el encargado de ingresar sus datos al sistema en la ventana de inicio. No obstante, los códigos de ciudades y horario de vuelos son datos que debe ingresar el programador.
  -

Las funcionalidades usadas son las siguientes:

- ***Fly***

Fly es la clase donde se almacenan todos los datos del transporte que maneja la aplicación, posee cinco atributos, availability, city, Price, chairs y hour. Estos son los datos primarios que se ocupan para realizar un viaje, puesto que esta es la clase que reúne todos estos datos, solamente posee los métodos getters y setters que llaman a sus distintas clases para verificar y editar información con sus respectivas restricciones.

- ***Destination***

Es la clase que se encarga de almacenar las ciudades y sus precios de viaje, codificados como enteros, posee dos métodos únicamente, los getters de city y Price respectivamente. La razón de que no tenga setters, es que no se podrá cambiar la ubicación de donde va dirigido el vuelo, como consecuencia tampoco cambiará su precio.

- ***Chairs***

Posee un número constante, el número de sillas que tiene el viaje y un arreglo de este mismo tamaño que representará las sillas, este arreglo será booleano para verificar qué silla está o no disponible. Posee dos métodos getter, uno para retornar el estado de todo el arreglo y otro para retornar uno en específico, hay un método adicional además del setter único.

  - *getAvailability:* Observará el estado de todas las sillas, si alguna de ella está disponible retornará true, sin embargo, si ninguna lo retorna false, y le da esta información a la clase Fly.

- ***Hour***

Está conformado por un array de dos posiciones, que almacena en la primer posición la hora, y en la segunda los minutos, contará con setter y getter. En el constructor, así como en el setter, se verificarán que los datos ingresados estén en el rango correcto de 24 horas para no tener una hora no lógica, posee también un método adicional

  - *DisplayHour:* Este método se encarga de imprimir la hora en la interfaz de usuario, teniendo el formato visualmente agradable EJ: 18:05

- ***Tree***

Es el encargado de organizar todas las reservas que se realizan en un día, cada una de las reservar contiene un número entero que será el identificador del nodo, se asegura que es único, y cada nodo contendrá la información del usuario y su respectivo vuelo.

Dicho en otras palabras, para realizar lo anterior primero se debe crear un objeto (se realiza en tiempo de ejecución) Tree; el cuál va a almacenar arreglos de números enteros. En este objeto es posible realizar las siguientes funciones:

  - *Insert:* Recibe el ID del usuario y lo agrega al árbol para poder ser editado posteriormente.
  - *Delete:* Recibe una reserva la busca en la lista y la elimina. Si no existe ninguna reserva en esa lista se muestra un mensaje "La lista está vacía, no es posible eliminar reservas". Si la reserva no existe en esa lista entonces se envía "Ésta reserva no se ha encontrado, no es posible eliminarla" y si se encontró "La reserva se ha eliminado exitosamente". .
  - *XOrder:* Muestra los ID del árbol de todas las reservas que se han realizado ese día y además un mensaje que dice "Las reservas encontradas son las siguientes:". Si no se ha hecho ninguna reserva ese día, entonces "La lista está vacía, no es posible mostrarla". No recibe ningún parámetro de entrada, X hace referencia al tipo de orden que se espera, inOrder, preOrder o postOrder.
  - *Find:* Recibe el ID del usuario y la raíz del árbol los busca en el árbol, y finalmente retorna el nodo en donde debería estar si no existe, o lo retorna de manera correcta si ya existe un nodo con ese ID.
  - *Rebalance:* este método se encarga de mantener el árbol balanceado mientras se agregan o quitan datos para mejorar la efectividad al momento de hacer búsquedas o hacer nuevas inserciones o deletes. Está complementada con funciones RebalanceRight, RebalanceLeft, RotateRight, RotateLeft.
  - *Next:* Mostrará el siguiente ID que exista con respecto a uno dado, no se implementa para uso del usuario, pero este ayuda para funciones de balanceo

IV. AVANCE EN LA IMPLEMENTACIÓN DE LA INTERFAZ DE USUARIO

La interfaz que permitirá al usuario interactuar con nuestro software está compuesto principalmente por tres ventanas:

La primera (página de inicio) solicita la información básica tanto del usuario como del viaje que desea realizar. Los datos del usuario son el nombre y el correo y; los del viaje son: la fecha de salida, el origen y el destino, esto con el fin de realizar un primer filtro para saber que vuelos se ofertan con estos requisitos. Además, se muestra la opción de "Ver reservas" en la que podrá observar o cancelar las reservas. A continuación se muestra el prototipo desarrollado en Balsamiq versus el avance de implementación.

Figura 2. Mockup Página de inicio

Para el desarrollo de la ventana principal, se aprovechó la herramienta Swing JFrame de Java y se distribuyeron los distintos paneles en una cuadrícula de tamaño 450x500. Esta ventana, tiene los botones de ver reservas y buscar que fueron implementados con ActionListener que llevan a la tercera y segunda ventana respectivamente. Además tiene tres espacios para ingresar la información solicitada y dos para seleccionar la ciudad de origen y destino.

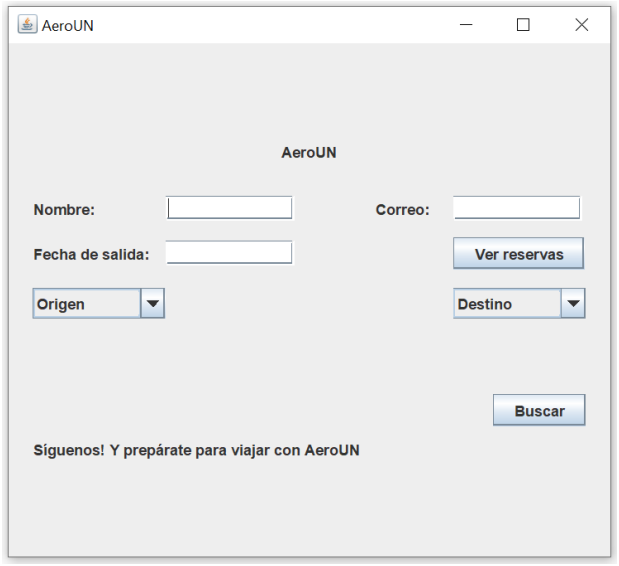


Figura 3. Avance página de inicio

Después de ingresar dichos datos, el software mostrará una ventana donde se efectuarán otros filtros para encontrar el vuelo ideal. En la opción "Tú vuelo más" se puede seleccionar si se desea que el vuelo sea más económico (precio), más temprano o tarde (hora de salida), entre otros. También, se puede encontrar la opción para seleccionar el número de pasajeros. Por último, la interfaz informará al usuario el número del vuelo, la hora de salida del lugar de origen, la hora estimada de llegada al destino, las sillas disponibles, el valor del tiquete y la opción para reservar.

v.

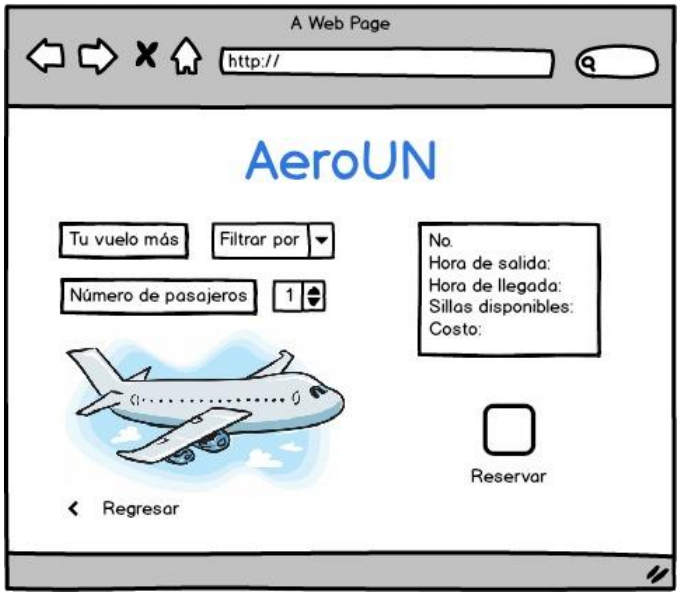


Figura 4. Filtros e información del vuelo

Cabe resaltar, que para la implementación de la ventana secundaria, se utilizó la herramienta JPanel que permite trabajar en un mismo frame pero alternando los paneles que se muestran.

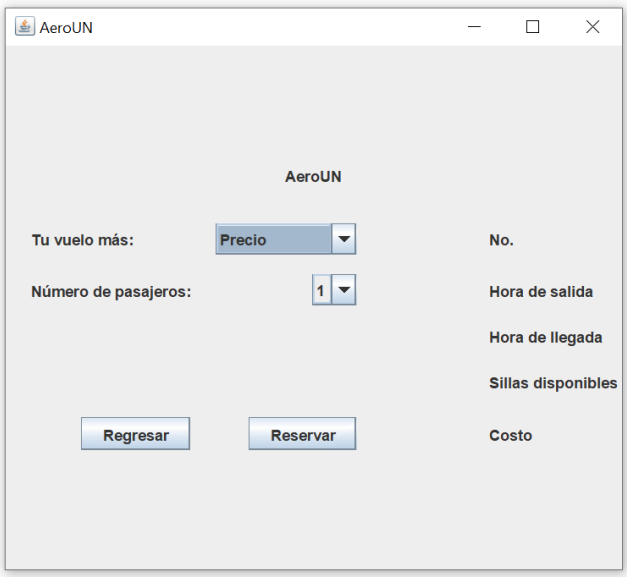


Figura 5. Implementación filtros e información del vuelo

Al finalizar la reserva se mostrará un aviso "Tú reserva está lista" y los datos finales del viaje, brindando la posibilidad de cancelarla o imprimirla. Adicionalmente, se tendrá la opción de regresar a la página anterior para modificar algún dato.

Figura 6. Información final de la reserva

En una segunda iteración del prototipo se decidió que no se realizaría la función imprimir ya que la información referente al vuelo ya aparece en pantalla. En lugar de ello, se optó por añadir un botón que le permita al usuario retornar a la página principal en caso de que quiera modificar o realizar alguna reserva.

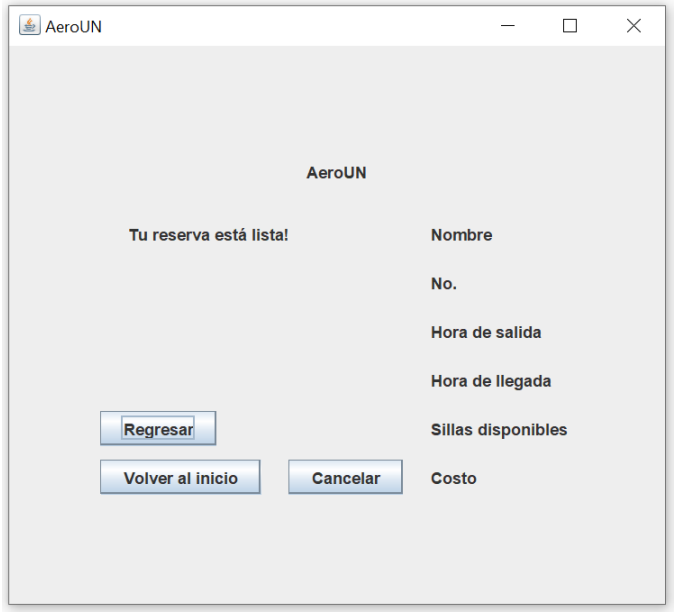


Figura 7. Implementación información final de la reserva

En el siguiente diagrama se puede observar cómo sería el recorrido del usuario a través de la interfaz de reserva de tiquetes aéreos.

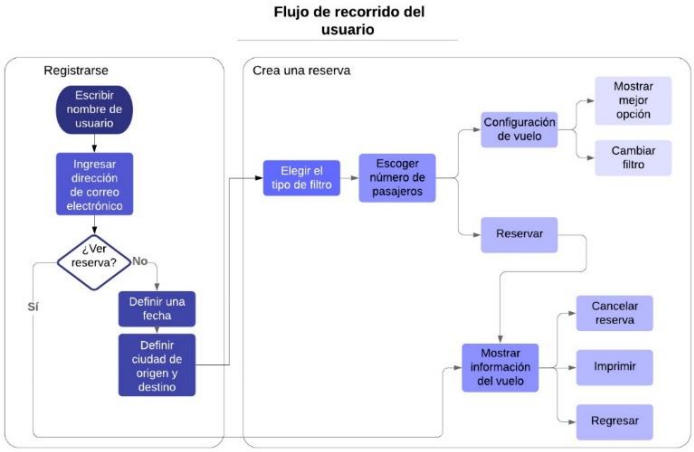


Figura 8. Diagrama de flujo. Ver Anexo B

VI. Entornos de desarrollo y de operación

El software que se va a desarrollar se realizará por medio del lenguaje de programación Java usando el entorno de desarrollo de Eclipse y elaborado para el sistema operativo Windows.

Para el manejo de versiones de dicho software se usará la plataforma de desarrollo colaborativo GitHub, en el cuál se crearán varias ramas donde lo integrantes subirán sus avances para luego unirlas. Los diagramas de lenguaje unificado de modelado (diagramas UML) y diagramas de flujo se harán a través del software Lucidchart y el diseño de la interfaz y los Mockups mediante Balsamiq.

El hardware para realizar las pruebas de dicho software presenta las siguientes características:

Procesador	Intel Core i3 - 2350
Velocidad de reloj	2,3 GHz
Número de núcleos	2
Generación	2
Memoria RAM	8 GB
Disco	500 GB
Tipo de sistema	x64

Tabla 1. Características del hardware

VII. DESCRIPCIÓN DEL PROTOTIPO DE SOFTWARE

En este segundo prototipo se cambió la lista doblemente encadenada que se usaba para almacenar las reservas realizadas durante el día usado arreglos con cuatro espacios por un árbol. Esto se debió a que al buscar un dato en una estructura de árbol el tiempo de ejecución es menor al de una lista (como se muestra en la siguiente sección); además, éste nos permite organizar la información de distintas maneras con lo cual se pueden generar filtros, cómo lo son el de precio, hora y disponibilidad de sillas.

En el siguiente gráfico se muestra en un diagrama UML como es el comportamiento y donde se usan las estructuras.

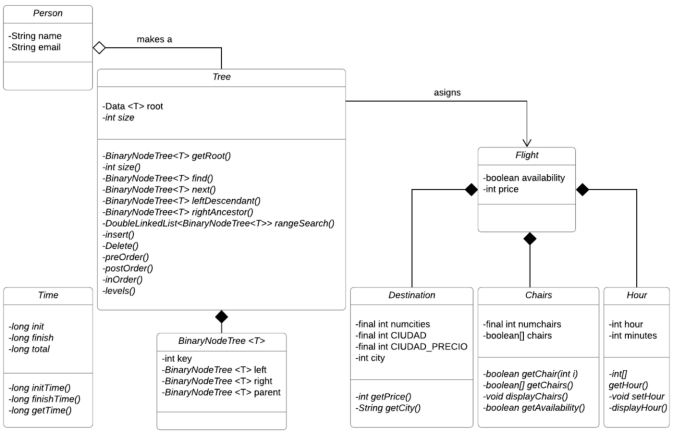


Figura 9. Diagrama de flujo. Ver Anexo C

VIII. PRUEBAS DEL PROTOTIPO

Para esta entrega se ha realizado una prueba inicial del prototipo, puesto que no se dispone de una base de datos organizada aún se añade al tiempo, crear y añadir el objeto, en este caso se mostrará los resultados en una tabla.

Cantidad de datos	Lista D enlazada	Árbol
1.000	3,9 ± 1	8,8 ± 6,5
10.000	11,7 ± 3,5	31,7 ± 24,5
100.000	42,6 ± 10	133,9 ± 16,5
1.000.000	282,5 ± 67	1597,2 ± 238,5
10.000.000	14.754,8 ± 625,5	43.962,1 ± 1.841,5

Tabla 2. Tiempos de ejecución para inserción

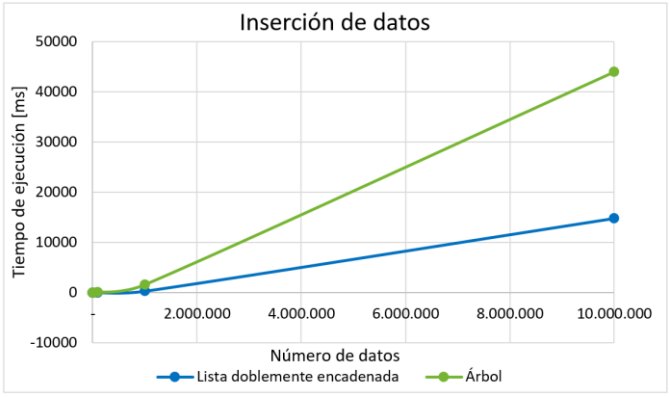


Figura 10. Gráfico comparativo de inserción de datos

Cantidad de datos	Lista D enlazada	Árbol
1.000	0,7 ± 0,5	0
10.000	2,1 ± 0,5	0
100.000	10,1 ± 1,5	0
1.000.000	18,6 ± 2	0
10.000.000	131,3 ± 10,5	0

Tabla 11. Tiempos de ejecución para búsqueda dato aleatorio

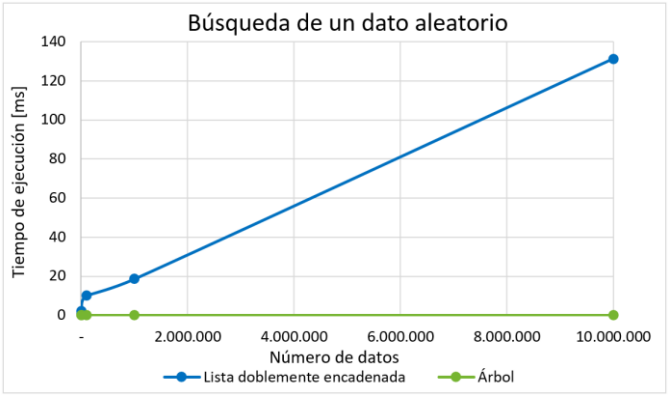


Figura 12. Gráfico comparativo de búsqueda de un dato aleatorio

Cantidad de datos	Lista D enlazada	Árbol
1.000	0,1 ± 0,5	0
10.000	2,2 ± 1	0
100.000	10,4 ± 4,5	0
1.000.000	22,6 ± 7	0
10.000.000	136,8 ± 8,7	0

Tabla 4. Tiempos de ejecución para eliminación dato

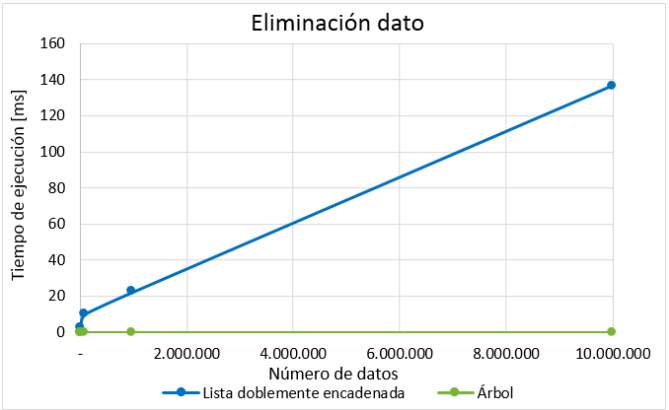


Figura 13. Gráfico comparativo de eliminación dato

Para obtener estos promedios de tiempo, se registró una muestra de diez datos del programa en ejecución y se realizó hasta 100 millones de datos, ya que con el hardware disponible ha desbordado la memoria RAM; el tiempo ascendió a casi 4 minutos tomado con cronómetro físico.

En las funcionalidades encargadas de la inserción de los datos, *PushFront* e *Insert* para la lista doblemente enlazada y el árbol, respectivamente, se puede observar que desde 1.000.000 de datos a 10.000.000 presentan un comportamiento lineal. Si sólo se tomara de 0 a 1.000.000 de datos se vería un comportamiento similar, por lo tanto, se dice que es O(n). Asimismo se puede percibir que para añadir datos a una estructura es preferible la opción de la lista enlazada ya que mientras esta tarda aproximadamente 14.754,8 ± 625,5 ms, el árbol tarda 43.962,1 ± 1.841,5 ms.

De igual manera, para la acción de búsqueda de un dato aleatorio, la mejor opción es la implementación en árbol (*Find*), debido a que en el peor caso no tendría que recorrer todos los datos, sino que iría filtrando cada que llegue a una nueva rama; además, tarda menos de 1 ms (por eso se observa ceros en las tablas) lo que genera que sea constante O(1).

Por último, la funcionalidad de borrar (*Delete*) es constante en el árbol debido a que no se tiene en cuenta la parte de buscar el elemento, sino que sólo se realiza un cambio de apuntadores. En el caso de *delete* para la lista encadenada si realiza la acción de búsqueda y eliminación en el mismo método, por esta razón los tiempos son similares, y por consiguiente, es lineal O(n).

Nombre de la funcionalidad	Tipo(s) de estructura de datos	Cantidad de datos probados	Análisis realizado (Notación Big O)	Tiempos de ejecución [ms]
PushFront (Inserción de datos)	Lista doblemente encadenada	1.000 – 10.000.000	O (n)	3,9 ± 1 - 14.754,8 ± 625,5
Insert (Inserción de datos)	Árbol		O (n)	8,8 ± 6,5 - 43.962,1 ± 1.841,5
find (Búsqueda de un dato aleatorio)	Lista doblemente encadenada		O (n)	0,7 ± 0,5 - 131,3 ± 10,5
Find (Búsqueda de un dato aleatorio)	Árbol		O (1)	0 - 0
delete (Eliminación dato)	Lista doblemente encadenada		O (n)	0,1 ± 0,5 - 136,8 ± 8,7
Delete (Eliminación dato)	Árbol		O (1)	0 - 0

Tabla 4. Tabla comparativa de las pruebas realizadas.

IX. DIFICULTADES Y LECCIONES APRENDIDAS

Hasta el momento la dificultad más grande que se ha enfrentado ha sido la de armonizar los esfuerzos de los integrantes en torno al proyecto, ya que al principio se tuvo problemas de comunicación. En particular, a pesar del esfuerzo puesto por cada uno de los integrantes del equipo, no se tuvo claridad de la forma de llevar a cabo e integrar las distintas clases por lo que se tuvieron situaciones en que dos integrantes realizaron un mismo trabajo o incluso un trabajo diametralmente opuesto. Para solucionar esto, antes de comenzar la implementación de la entrega intermedia, se realizó una reunión extra-clase en la que se discutió a profundidad cómo y en qué clases se utilizarían los árboles y se asignaron tareas específicas de tal forma que se logró un resultado más satisfactorio que en la primera entrega.

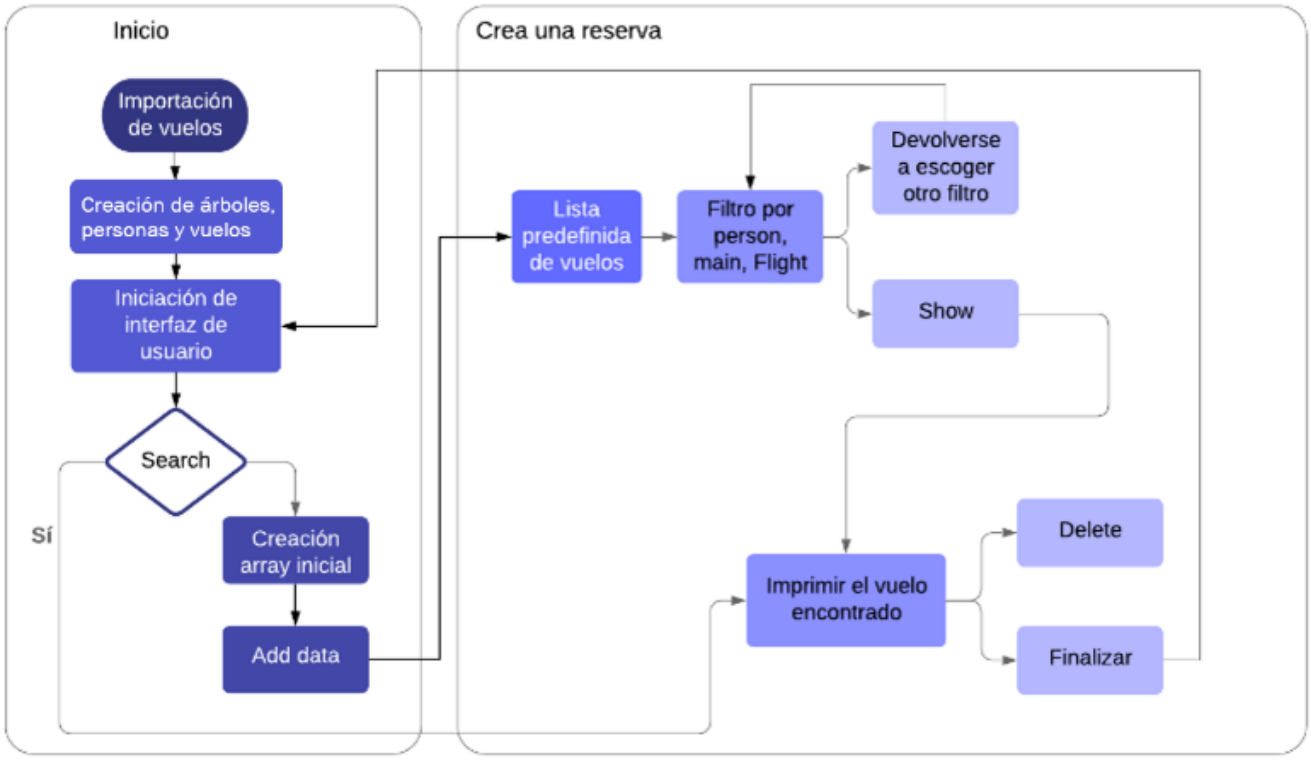
Por otro lado, en cuanto a problemas técnicos, se tuvo dificultades para realizar la acción del balanceo del árbol puesto que no se comprendía cómo realizar su implementación en código. Al respecto, se revisaron nuevamente las diapositivas provistas desde la plataforma coursera, se discutió el problema en el grupo y se logró implementar éste método adecuadamente.

Asimismo, se tuvo dificultades en la implementación de la interfaz ya que se vio la materia de programación orientada a objetos hace varios semestres. En este sentido, fue necesario recapitular el tema de JFrame y JPanel y entender sus diferencias y funcionalidades. Sin embargo, una vez dedicado el tiempo suficiente se pudo aprovechar los botones, listas, y cajas de texto en busca de emular lo propuesto como mockups.

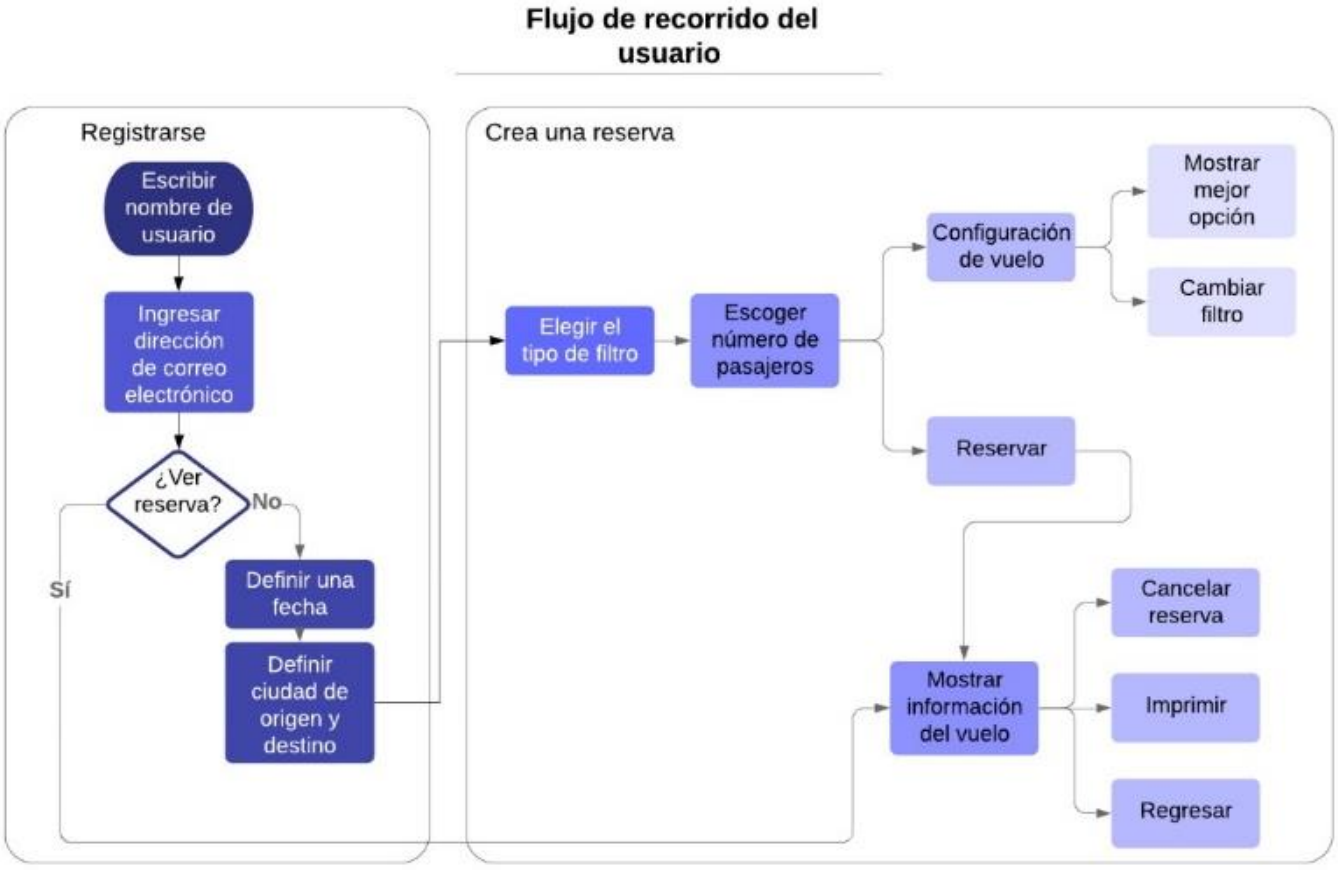
En conclusión, se podría decir que la lección aprendida más importante es que a pesar de las dificultades, es esencial buscar maneras de trabajar mejor en equipo y de apoyarse en los compañeros para poder lograr más fácilmente los objetivos propuestos.

ANEXOS

Anexo A. Diagrama de flujo del sistema.



Anexo B. Diagrama de flujo del usuario.



Anexo C. Diagrama UML del software.